CS148 Homework 5

Homework Due: Jul 31st **(Thursday)** at 2PM - 5PM (In-Person) or 7PM - 8PM (Remote)

0.1 Assignment Format

PLEASE READ: This assignment is split into two distinct parts:

- The first 3 `TODO`s are all coding, and they are prefaced with directions on what you need to code up. These TODOs cover the topic of sampling area lights and global illumination (for color bleeding).
- The final `TODO` is all done in the Blender GUI. You get to choose two of the following topics:
  - fluid simulation
  - cloth simulation
  - depth of field
  - motion blur
  - volume rendering

  to apply to your custom scene (for your final project). This TODO section is sparse and leaves the instruction to the Blender tutorials in the rest of this handout.

0.2 Collaboration Policy and Office Hours

All policies are the same as they were for HW1. See the HW1 document for details.

Quiz 4

You will be randomly asked one of these questions. We won't cover them until Tuesday.

- What concerns may come up if we were to use Explicit (Forward) Euler to evaluate the equations of motion when updating the state of a physical system? How do these issues compare to those of Implicit (Backward) Euler?
- What is the difference between the Lagrangian fluid simulation approach vs. the Eulerian fluid simulation approach? What is the advantage of combining both approaches?
- How are springs relevant to cloth simulation? Describe the three types of springs that we need to construct the simplest cloth model.
- How can we force a collision in our physical simulation to be inelastic vs. completely elastic? How can we simplify the problem of modeling a 3D collision into a 1D problem?
- Describe one technique for generating the inbetween frames of a scene in motion given a set of manually crafted keyframes. Given a bunch of frames of a scene in motion, how do we ray trace motion blur?

1 Assignment

1.1 Area Lighting & Global Illumination

Your task this week is to adapt the simple ray tracer from HW3 from using point lights and direct illumination to using area lights and global illumination for more natural lighting. To demonstrate the effect, we will apply the simple ray tracer to the classic Cornell box scene shown below.

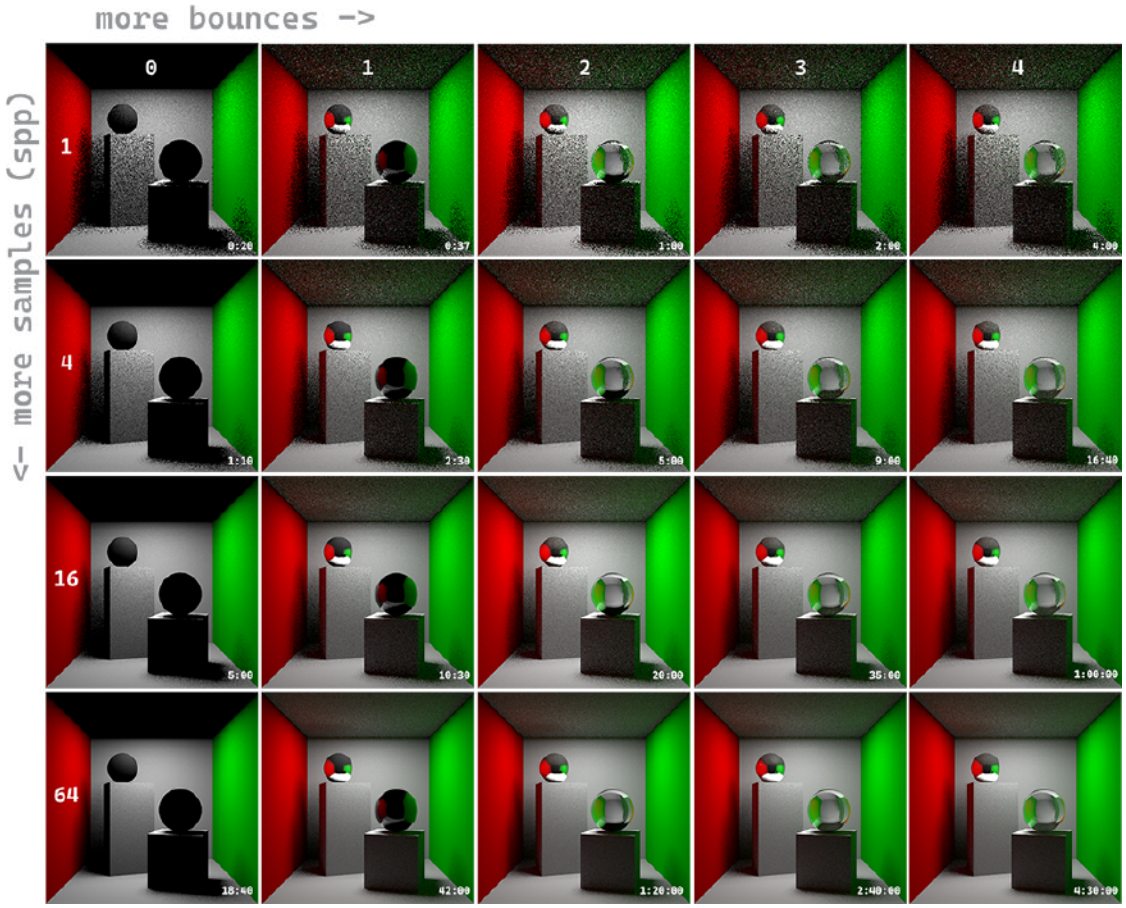
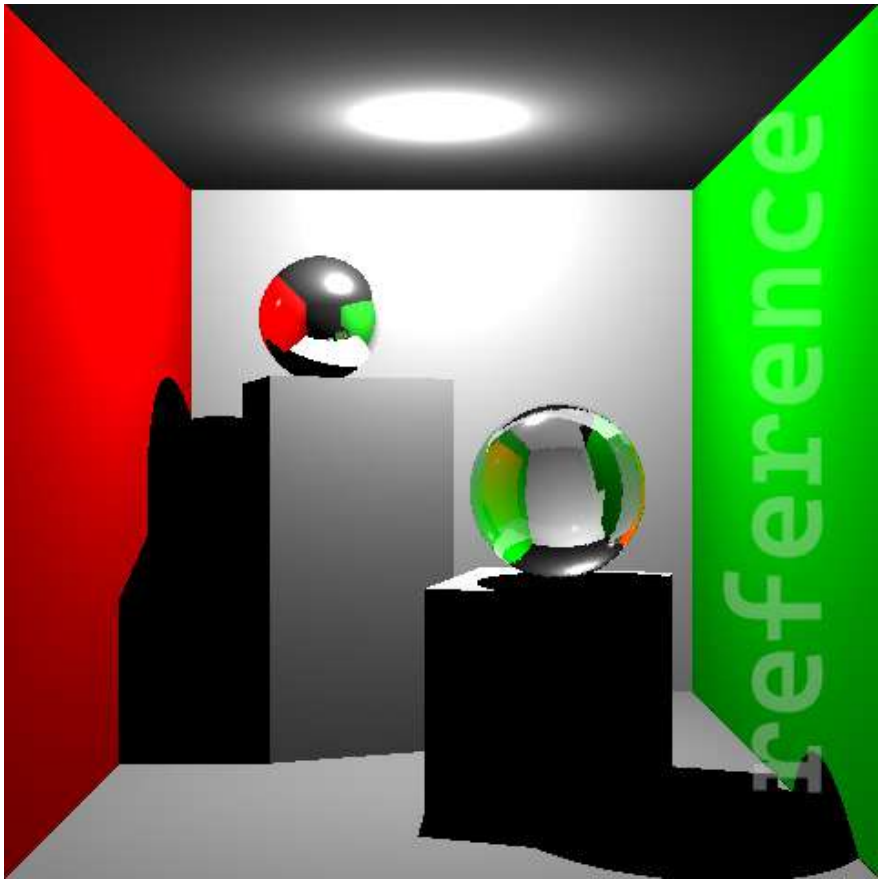


Figure 1: The Cornell box scene rendered using the completed HW code; timed on a i9-9900X CPU @ 3.50GHz (single-threaded)

We've already created the scene for you in Blender and have set up the code infrastructure to access information from the scene using Blender's Python API. **You can find the scene and code all** in this .blend file if you are using any Blender version before 4.4. If you are using Blender 4.4 or above, then use in this .blend file.

To run the code, you will need to run the `simpleRT_UIpanels` script first every time you launch Blender (like in HW3) before running `simpleRT_plugin`. You should try launching Blender from the command line as you did in HW3. The starter code comes with a function that prints out the estimated wait time to the command line for how long the render will take to finish. Rendering the scene with the starter code should give you the following image:



1.1.1 TODO 1: Implement an Area Light (1 pt)

We will go over step-by-step the process of implementing an area light in code.

1. We first need to check if a light is an area light (as opposed to a point light) when iterating over all the lights in the scene for our ray tracing. We can do so using the condition: `light.data.type == "AREA"`.

   Add an if statement within the loop over the lights like so:

   ```
   for light in lights:
       light_color = ... # don't modify
       light_loc = ... # don't modify

       # ADD CODE FOR AREA LIGHT HERE
       if light.data.type == "AREA":
           ... # your code

       light_vec = ... # don't modify
   ```

   If the light is an area light, then we proceed with the following steps to update the light color variable, `light_color`, as well as the light location variable, `light_loc`.

2. Calculate the normal vector for the area light in world space. Area lights are surface patches

and thus have normals that we use to determine their tilt angle with respect to the object(s). You know that the tilt angle affects the strength of the light.

We calculate the light normal by first defining it in the light's local space (basically the light's "object space" if we pretend the light were an object). Let the area light be emitting downwards (i.e. in negative z) in its own local space, thus initialize:
`light_normal = Vector((0, 0, -1))`
Then, we need to transform this normal vector for the light into the global world space. Conveniently, the light data structure already has the transform we need stored as a member variable, `rotation_euler`, already computed via Blender. So we just need to call:
`light_normal.rotate(light.rotation_euler)`.

3. Update the light color based on the tilt angle between the area light and object. To do so, we need to multiply `light_color` by a dot product between the light normal and the direction **FROM** the light **TO** the hit location. Remember to normalize the vectors since we are relying on them for a dot product, and be mindful of the vector directions! If the dot product is negative, then we can set `light_color` to zeros, since it means the object is behind the area light. Unlike point lights, area lights only emit forwards in one direction.

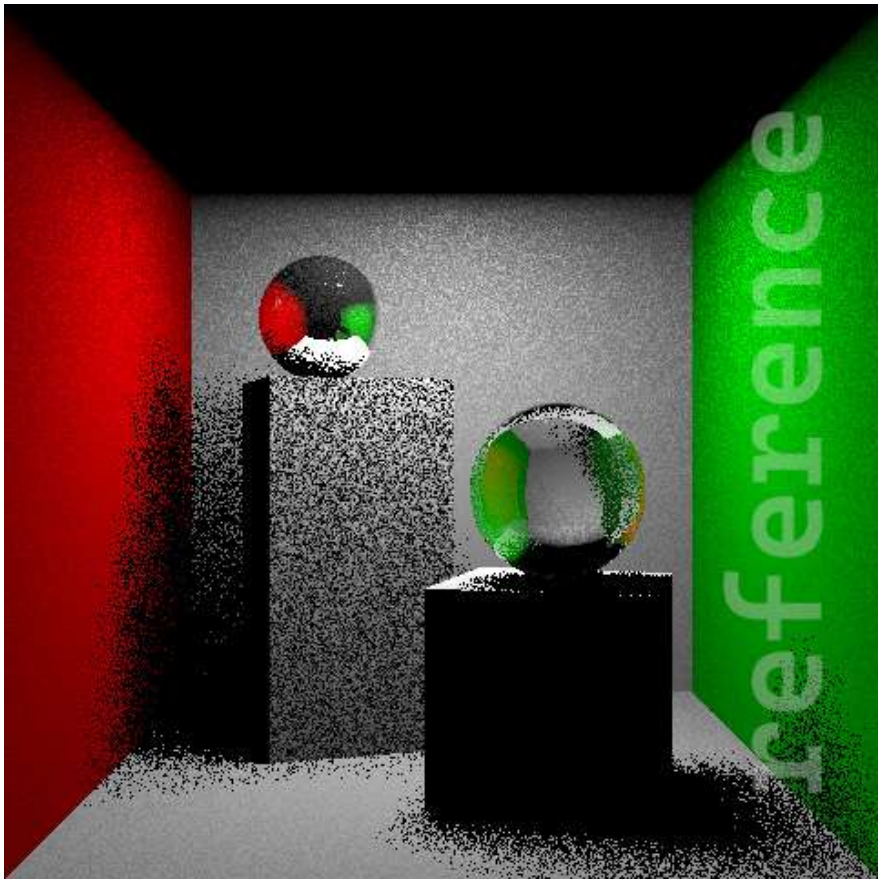
4. Calculate the point on the area light disk from which we will be emitting light. Conceptually, you can think of a (disk-shaped) area light as a collection of point lights that only shine in 1 direction, clustered together into a shape (of a disk). For this step, we will (randomly) pick one of the point lights in this collection to emit light from. We will worry about the other point lights in our collection later. This step will be done in the local space of the light.

   Since the light is in the shape of a disk, we can randomly sample this area by parameterizing the space such that each point has some distance to the disk center r and angle θ (as in polar coordinates). We uniformly sample r from 0 to 1, and θ from 0 to 2π. You can use the Python function `np.random.rand()` to generate random samples from a uniform distribution over $[0, 1)$.

   After we obtain our uniform samples, we can compute the emit location in Cartesian coordinates as $[x, y, z] = [\sqrt{r}\cos(\theta), \sqrt{r}\sin(\theta), 0]$. From here, we scale the coordinates with the radius of the disk, which is stored in the light data structure member variable: `light.data.size/2` to get the final sample coordinates. Save this as a new variable.

5. The previous step computes the light emit location in the light's local space, so we need to transform it into the global world space. Again, the transform that we need is conveniently already stored in the light data structure as a member variable: `light.matrix_world`. This stores the matrix that transforms a point from the light's local space to world space. You simply just need to apply the matrix appropriately to your computed coordinate in the previous step. Set the final result to the `light_loc` variable, thus modifying it appropriately for use later in the code.

After finishing the above steps, your render should look similar to the following (but might not match exactly because of random sampling). **Please save this render at 480x480 100% resolution with a depth of 3 for grading.**



1.1.2 TODO 2: Sampling the Area Light (0.5 pt)

You may have noticed that implementing the area lights introduced a lot of noise in the shadows. This is because we only emitted light from one point on the area light disk for each area light. To get more accurate lighting with area lights, we need to emit light from multiple points on the disk for each area light, then average over the results. This process is called sampling.

1. First, look for the `render(self, depsgraph)` function definition. Notice how the `self.samples` variable is set to 1. This is telling the code to do only 1 ray trace pass through the whole image. Replace the 1 with the code in comments to its right. This sets the samples variable to whatever number you enter into the `samples` field of the `Render Properties` tab in the Properties Editor.

   The sample number is basically how many ray trace passes we will do. If we only do 1 ray trace pass, then that means we only use 1 random point on each area light disk for lighting and call it a day. If we were to do 2 samples, then we would ray trace 2 times, thus using 2 random points on each area light disk instead. For 4 samples, we would ray trace 4 times, using 4 random points on each area light disk; and so on.

2. Now take a look at the `RT_render_scene` function. Notice the triple loop that first loops over the number of samples, then the height of the image, then the width of the image. Within the loop, notice the call to the ray tracing function that you implemented in HW3: `RT_trace_ray`. Does it make sense to you now how the sample number is the "number of ray trace passes" that we do?

Currently however, the code is not equipped to handle multiple ray trace passes. Look for the line:

`buf[y, x, 0:3] = ...`

This variable is short for "buffer", which refers to the data structure that we use to store our image (which is simply an array of 2D arrays of pixels, one 2D array for each color channel: r, g, b, and alpha transparency). Recall that this is all within a loop over the height of the image, then a loop over the width of the image. This double loop is looping over each pixel in the image. Notice then how we are storing the r, g, and b color computation from our ray tracing function for each pixel directly to this buffer variable via the above line of code. This effectively has each iteration of the outermost sample loop overwrite the last by writing directly to the buffer, thus only the last sample iteration will matter.

We want to instead average our results across all the samples. To do so, you need to create a temporary buffer variable, i.e. call it the sample buffer or `sbuf` for short. Declare this variable outside the triple loop as a 3D array with the same height, width, and 3 color channels as the buffer variable (ignore the alpha transparency channel here):

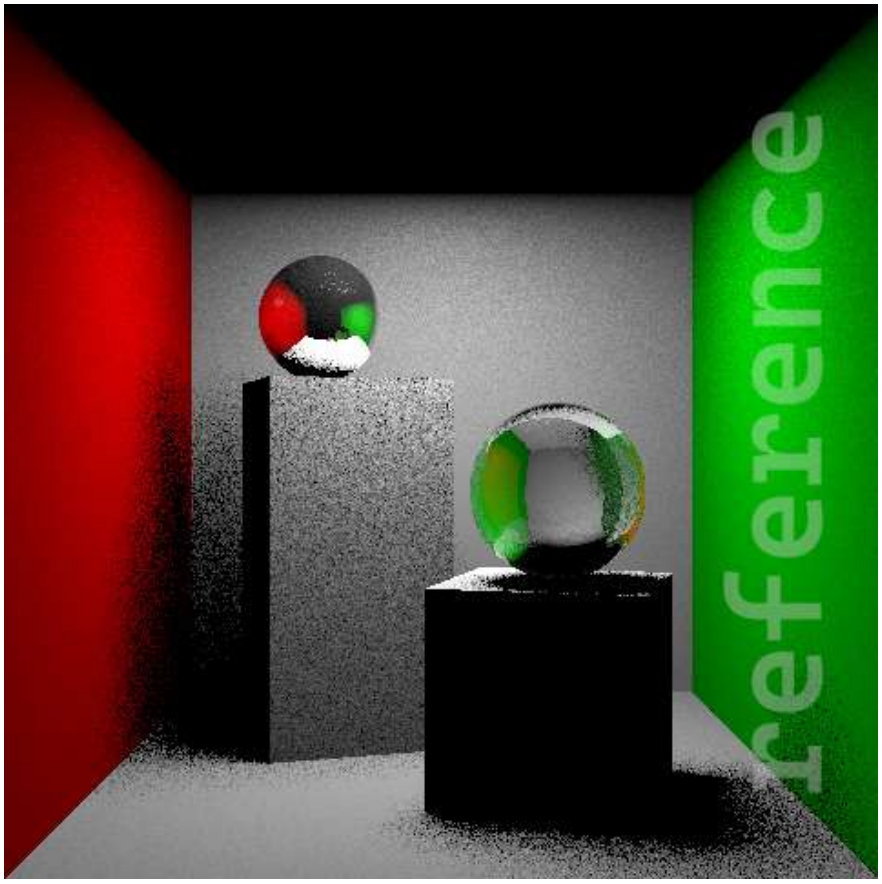
`sbuf = np.zeros((height, width, 3))`

From here, you want to **ADD** (as in accumulate) the results of the ray trace function to the sample buffer instead of the actual buffer within the loop.

Still within the loop, We can then get the appropriate image for the current sample count by dividing the data stored in the sample buffer by the current sample count `s + 1` (Python self-check: why the +1?). This gets us our desired average across all current samples in the loop so far. Finally, you can set the result of your computation to the original buffer:

`buf[y, x, 0:3] = # your result`

After finishing the above steps, your render should look similar to the following (but might not match exactly because of random sampling). **Please save this render at 480x480 100% resolution with 4 samples and a depth of 3 for grading.**



1.1.3 TODO 3: Global Illumination (1.5 pt)

The images so far have all had "dead" black shadows, i.e. the colors from the red and green walls do not "bleed" onto the cubes. This is because we only have direct diffuse and specular, i.e. the diffuse and specular rays stop once they hit an object. In reality, the object receives light from not only light sources, but also from other objects in the scene. You have heard of this phenomenon as color bleeding from lecture.

To mimic real-world lighting, we need to add more bounces for the rays to achieve global illumination. We will do so for the diffuse rays in this HW by giving them recursive bounces to implement what we call indirect diffuse lighting. Essentially, we will add extra steps to the computation of the diffuse component in the Blinn-Phong BRDF. We will ignore global illumination for the specular component to keep the assignment within reasonable length.

You can write this code right before the ambient computation in `RT_trace_ray`.

1. First, check if `depth > 0`. Similar to reflection and transmission rays, we only shoot recursive rays when the recursive depth is greater than 0. Only proceed with the next steps if this true.

2. For the purposes of computing the recursive ray direction, we need to first establish a local coordinate system with the normal vector at the intersection point as the Z-axis. That is, we need to find a pair of x and y axes so that the z-axis becomes `hit_norm`.

   For the x-axis, we will make it a unit vector orthogonal to the normal. Start by initializing this vector to an initial guess of (0,0,1). Now, if (0,0,1) is too close to the normal, then change it to (0,1,0) instead. Two vectors are "too close" if they are almost parallel, i.e. the

magnitude of their dot product is close to 1. (Self-check: do you know why? Ask in office hours if you're not sure!)

We then compute the normal-direction component of x, which can be computed as $x \cdot n * n$ where x and n are the x and normal vectors respectively. Essentially, this is just a dot product of the x and normal vectors, then a component-wise multiplication with the normal vector. Subtract the result from the x vector to make it orthogonal (aka perpendicular) to the normal, then normalize x. Some of you may recognize this process as the Gram Schmit orthogonalization technique from linear algebra.

The y vector can be obtained via the cross product of the x vector and the normal. Python has a `.cross()` function.

3. Also for the purposes of computing the recursive ray direction, we need to sample a hemisphere oriented at (0, 0, 1). Imagine the top half of a sphere centered at the origin of the xyz-axis, where up is positive z. We want to randomly pick a vector direction along this hemisphere.

   Mathematically, we can express a point on the hemisphere as $[\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta)]$, with $\theta \in [0, \pi/2), \phi \in [0, 2\pi)$. We uniformly sample on the hemisphere surface by making $\cos(\theta)$ a uniform variable between 0 and 1.

   Computationally, we do this by first creating two random variables r_1 and r_2 between 0 and 1, and let r_1 be $\cos(\theta)$ and $r_2 * 2 * \pi$ be ϕ. From here, you can simply plug these values appropriately into the above formula for a point on the hemisphere (remember that $\sin^2(\theta) + \cos^2(\theta) = 1$). Since the hemisphere is centered at (0,0,0), this point is also a ray direction. (Self-Check: do you know why? Ask in office hours if you're not sure!)

4. The ray direction that you computed from the previous step is within some abstract local space of a hemisphere, and thus needs to be transformed into world space to actually be used. To do this, we use the coordinate system computed in Step 2 to determine a matrix transform. This transform will take the ray from this abstract hemisphere and place it alongside the normal of our object in world space.

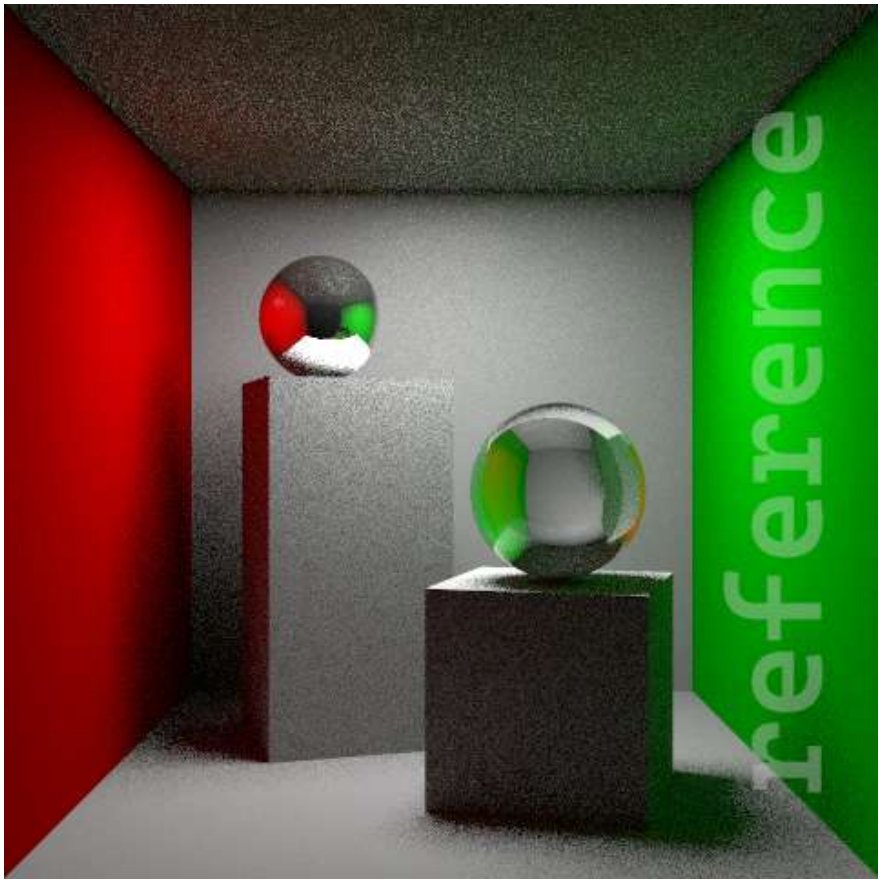
   Let `x, y, n` be the coordinate system you computed in Step 2. Then to form the necessary matrix transform in Python, you can do:

   ```
   mat_transform = Matrix()
   mat_transform[0][0:3] = x
   mat_transform[1][0:3] = y
   mat_transform[2][0:3] = n
   mat_transform.transpose()
   ```

   Use this matrix to transform your ray result from Step 3.

5. Finally, recursively trace the ray from Step 4, remembering to take into account self-occlusion. Store the result as the indirect diffuse color. This color needs to be scaled by r_1 to account for face orientation as well as `diffuse_color` to account for absorption. Add the final result of these products to `color`.

After finishing the above steps, your render should look similar to the following. When you think you have your code correct, **render your final image at 480x480 100% resolution with 16 samples and a depth of 3. This may take up to an hour or more. Please save this render for grading.**



In case you are curious, this is how the scene from HW3 looks if rendered with the completed HW5 code. Notice the softer shadows and color bleeding. You do not need to generate this image yourself.

1.2 TODO 4: Render a prototype of your final project scene

In lecture, we talked about simulation for both fluids and cloth. We also talked about depth of field and motion blur, and later in the class, we'll go over how to implement volume rendering. In this PDF, you'll find tutorials below on how to set up simple examples for all of the above.

For the purposes of grading, pick at least TWO of the above to include in a prototype of your final project scene. Make sure the two you choose are actually relevant for your project. For instance, we don't want to see a heavily motion blurred car randomly in your scene for the sake of this assignment; rather, if you do add motion blur to your scene, then it has to make sense for your project!

NOTE: While we haven't discussed the theory of how we model and implement volume rendering and will not until after this HW is due, you don't actually need to know those details to enable volume rendering in Blender. For now, if you are interested in using volume rendering for your final project, then take the time to experiment with the Blender GUI options. We will talk about what's going on under the hood later.

Show us: (2 pts) A ray traced image using Blender Cycles of a prototype of your final project scene with at least TWO of the following included within the scene: geometry from fluid simulation, geometry from cloth simulation, depth of field, motion blur, and volume rendering.

Whichever two (or more) you include, be ready to defend your decision on including them and explain why they'll be relevant/necessary for enhancing your final project.

That covers all the TODOs! Everything else in this document consists of instructional (Blender) tutorials for you to reference.

2 Creating Geometry via Fluid Simulation

Blender provides a relatively easy-to-use interface for generating workable meshes from fluid simulation(s). Let's try it out with a simple example. First, taking the default Blender scene:

- Scale the default cube to enlarge it. This large cube will become our fluid domain.
- Add a smaller cube mesh inside the default cube. This small cube will be our collision object.
- Add an UV sphere mesh above the smaller cube and inside the outer cube. This sphere will contain our liquid that gets simulated in our fluid domain.

It may help to switch to wireframe view in the Viewport toolbar when setting this up. See Figure 2 for a visual of the full setup.

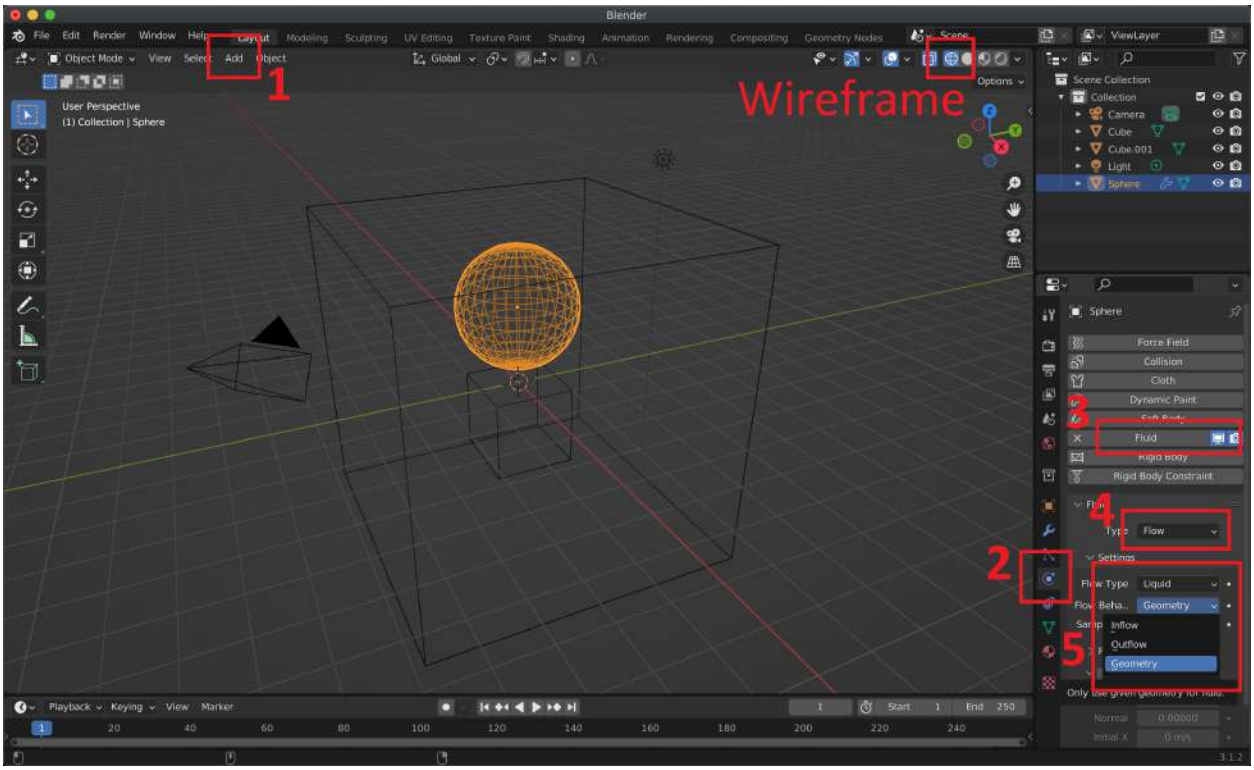


Figure 2

Start by selecting the sphere, then:

- Click on the **Physics Properties** panel of the Properties Editor (#2 in Figure 2) to access the physics options.
- Click on the **Fluid** option (#3 in Figure 2).
- Click on the **Type** dropdown menu and select **Flow** (#4 in Figure 2).
- Click on the **Flow Type** dropdown menu and select **Liquid** (#5 in Figure 2).
- Click on the **Flow Behavior** dropdown menu and select **Geometry** (#5 in Figure 2).

This sets up the sphere to be a giant collection of liquid particles. The **Geometry** option we set at the end tells Blender to have these liquid particles act as geometry when interacting with other objects in the scene (i.e. they will behave like actual mass when colliding with objects). In comparison, the **Inflow** and **Outflow** options will continuously add or delete fluid particles during the simulation, with the inflow simulating a running faucet, and the outflow simulating a drain with fluid disappearing upon interacting with the world.

Now, let's add a collision object. Select the smaller cube, then:

- Click on the **Physics Properties** panel of the Properties Editor.
- Click on the **Fluid** option.
- Click on the **Type** dropdown menu and select **Effector** (#1 in Figure 3).
- Click on the **Effector Type** dropdown menu and select **Collision** (#2 in Figure 3).

This sets up the cube to be an object that the sphere of fluid will collide with as it falls.

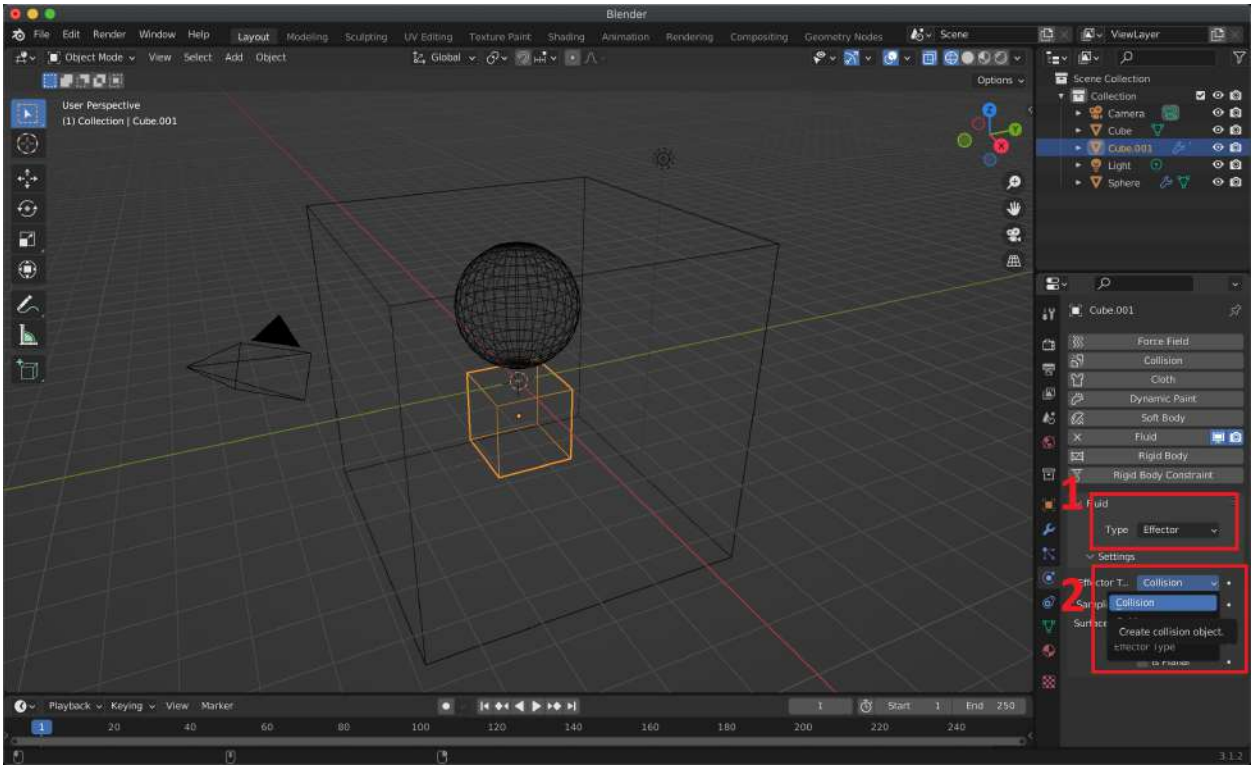


Figure 3

Finally, we need to set up the last cube in the scene. Select the large cube, then:

- Click on the **Physics Properties** panel of the Properties Editor.
- Click on the **Fluid** option.
- Click on the **Type** dropdown menu and select **Domain** to make this cube our fluid domain (#1 in Figure 4).

- Click on the **Domain Type** dropdown menu and select **Liquid** to tell Blender that this fluid domain will be used for simulating liquids (#2 in Figure 4).

This sets up the large cube to have a gravity field that will act on any liquid objects within. This includes the sphere of liquid, but not the collision cube. This means that when we run the simulation, gravity in our fluid domain will cause motion for the sphere of liquid and make it fall along the negative z-axis. But, the collision cube will stay in place. However, the sphere of liquid and the collision cube will interact, because we set the cube to be a collision object.

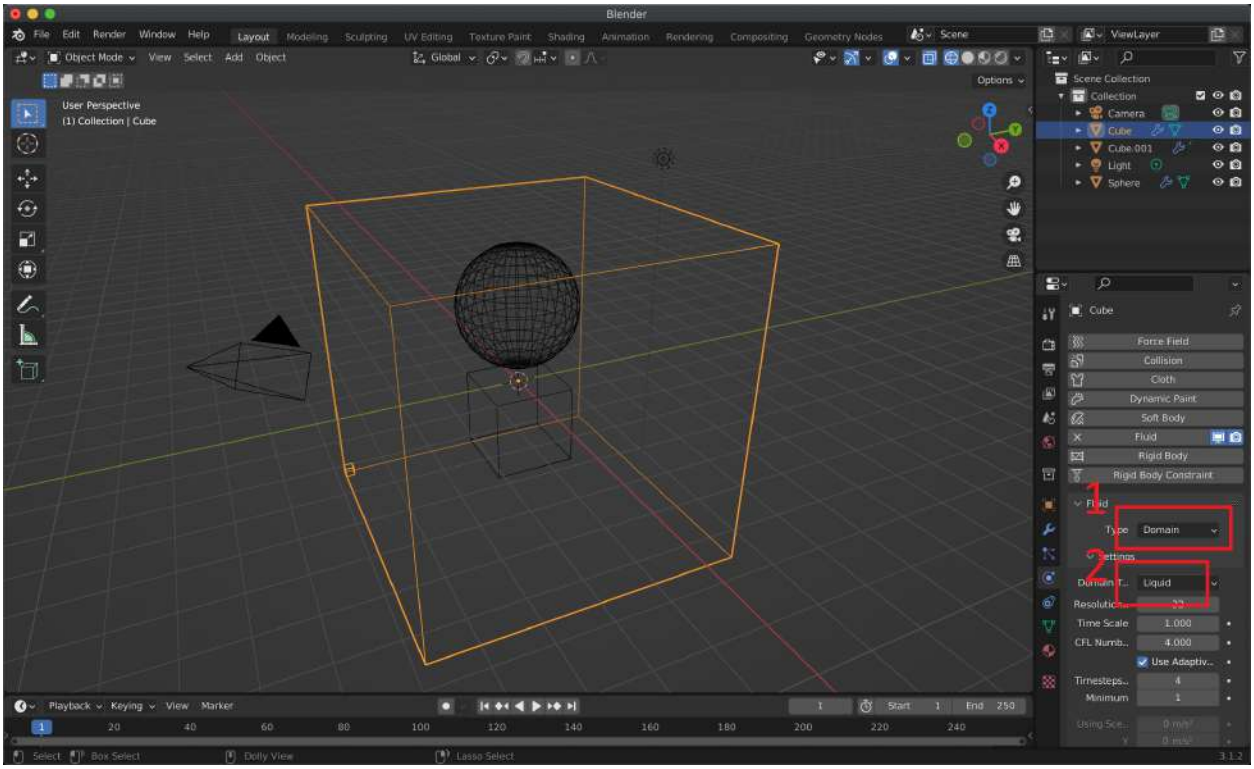


Figure 4

To actually run our fluid simulation, we need to set a few more parameters:

- Scroll down further in the **Physics Properties** panel of the large cube that we set to be our fluid domain.
- Set the **Cache** folder (#1 in Figure 5) to a folder where you want Blender to cache or temporarily save the files needed to model this fluid simulation.
- Set the **Type** for the simulation to **All** to tell Blender to generate geometry for all frames of the simulation (#2 in Figure 5).
- Check the **Mesh** checkbox to tell Blender to explicitly generate mesh geometry for the fluid simulation (#3 in Figure 5).
- Set the final **End Frame** of your fluid simulation in both the Properties editor and the **Timeline Editor** at the bottom of the Blender interface (#4 in Figure 5).

- Click **Bake All** when you're ready to generate your simulation (#5 in Figure 5). It may take a few seconds for Blender to finish computing. There should be a progress bar at the bottom.

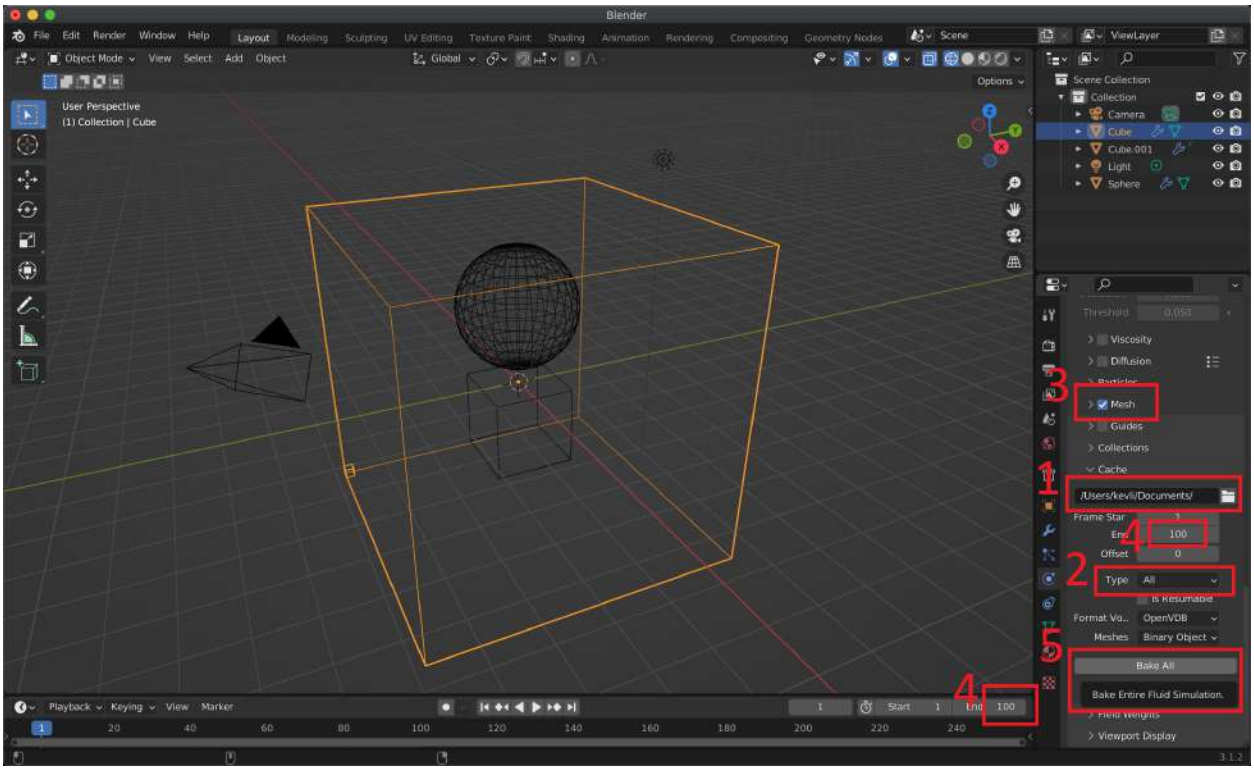


Figure 5

To see your fluid simulation in action, move the **Timeline Editor** at the bottom of the Blender interface up until you see the blue line clearly (see Figure 6). You'll see a timeline starting at 1 and ending at the **End Frame** you set earlier. You can scrub through the timeline by clicking and dragging the blue line to see the fluid simulation at different frames. For instance, Figure 6 shows us looking at the 50th frame of our simulation of 100 frames.

You may want to switch to solid view in the Viewport toolbar to get a better sense of the geometry of your fluid. If you want to redo your fluid simulation with a different set up, then you need to delete the cache files for the current simulation (see Figure 6) before modifying your scene.

When you find a frame in your simulation that you're happy with, you can export all the geometry in that frame as an .obj mesh. Simply use the **File → Export** option. Then, you can import that same .obj into your actual scene and work with it like any other geometry. For instance, in Figure 7, we show how we can transform the generated liquid mesh just like any other object. You can also add materials, textures, etc to your liquid mesh.

Warning: If you try to work off the Blender file from which you created the simulation, then you'll likely run into artifacts from the simulation set up. For instance, the fluid domain that we defined to encase our fluid simulation may end up getting rendered (which is something you don't want to happen)! So we recommend doing your simulation work in a separate file, and then exporting and importing the result as an .obj mesh to work with in a main scene file.

For more info on the parameters and options that Blender provides for fluid simulation, see their official manual. This Youtube tutorial on the "Quick Fluid" option in Blender may be useful, especially for finding an appropriate material for a water surface (in this case, the glass BSDF).

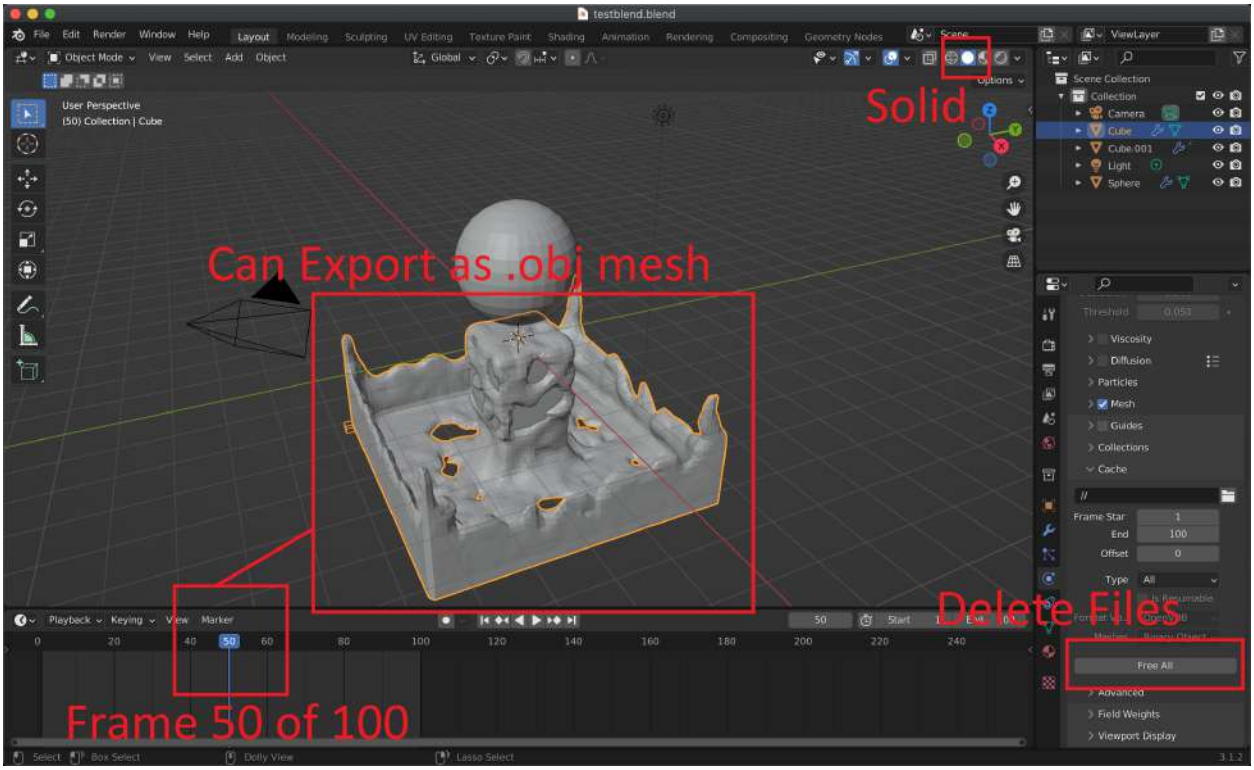


Figure 6

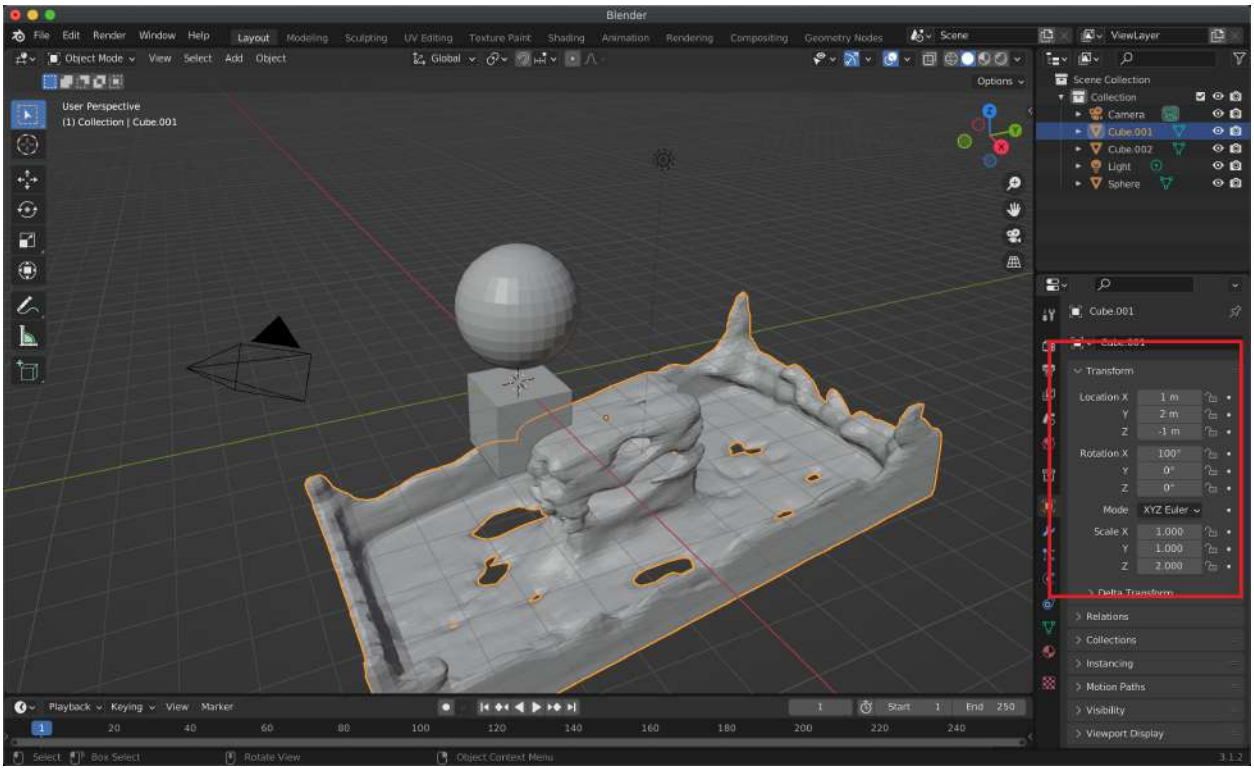


Figure 7

3 Creating Geometry via Cloth Simulation

Blender also provides a straightforward interface for deforming meshes with cloth simulation(s). Let's see with a simple example. Create a new scene, delete the default cube, and add a plane mesh plus the built-in **Monkey** mesh. Position the plane above the monkey head (see Figure 8) The plane mesh will become our cloth in our cloth simulation, and we'll have it collide with the monkey head.

Select the plane, then:

- Go into **Edit Mode**, then **Modifier Properties** in the Properties Editor (#1 in Figure 8).
- Click **Add Modifier** and add a **Subdivision Surface** modifier to allow us to subdivide the plane into a finer grid.
- In the modifier options, click **Simple** to use Blender's simple subdivision algorithm, which doesn't round out the edges of our plane (#2 in Figure 8).
- Right click the plane and click **Subdivide**. This will bring up a **Subdivide** control panel in the bottom left (#3 in Figure 8).
- Change the **Number of Cuts** to 50 to make the plane a 50x50 grid of squares.

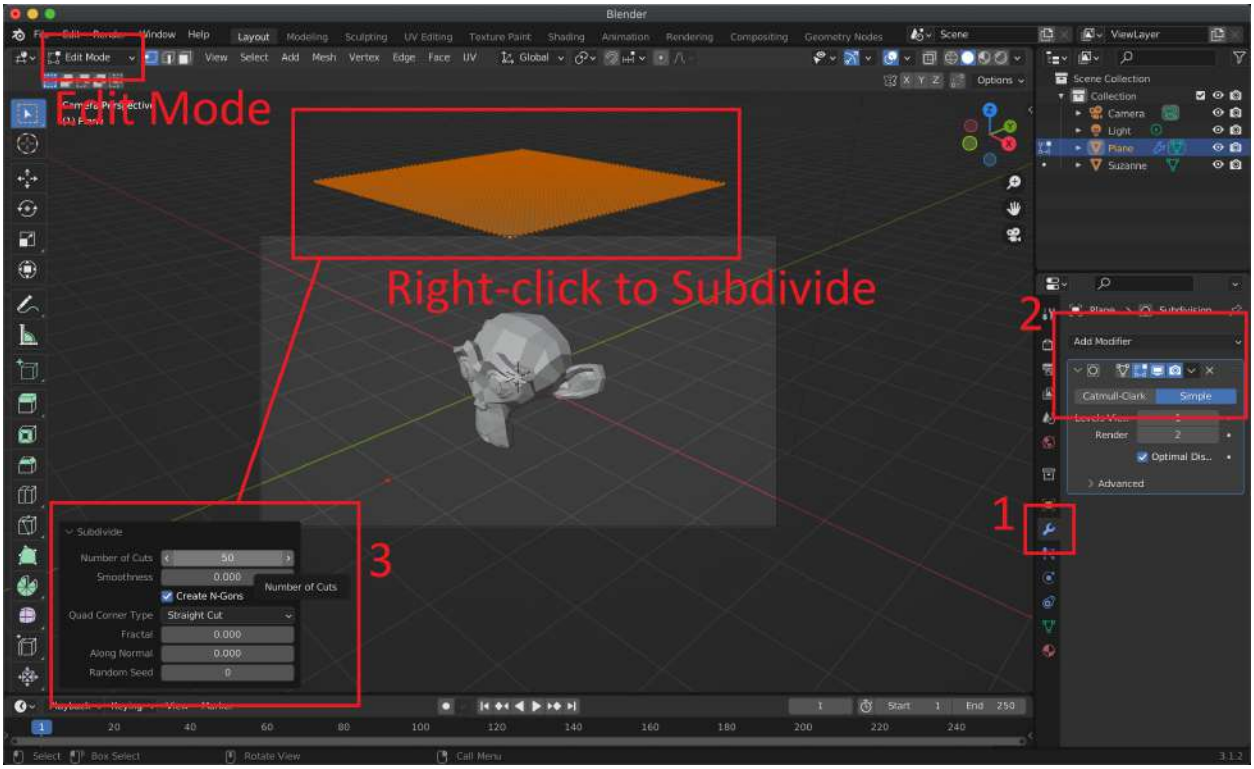


Figure 8

Now, we need to set the plane to be a cloth object and the monkey head to be a collision object.

- Select the plane (you want to go back to **Object Mode**), then click on the **Physics Properties** panel of the Properties Editor. Select the **Cloth** option. This tells Blender to treat our plane as a network of particles and springs to model the physics of a piece of cloth.
- Select the monkey head, then click on the **Physics Properties** panel of the Properties Editor. Select the **Collision** option. This tells Blender to have our monkey head interact with any objects that collide with it.

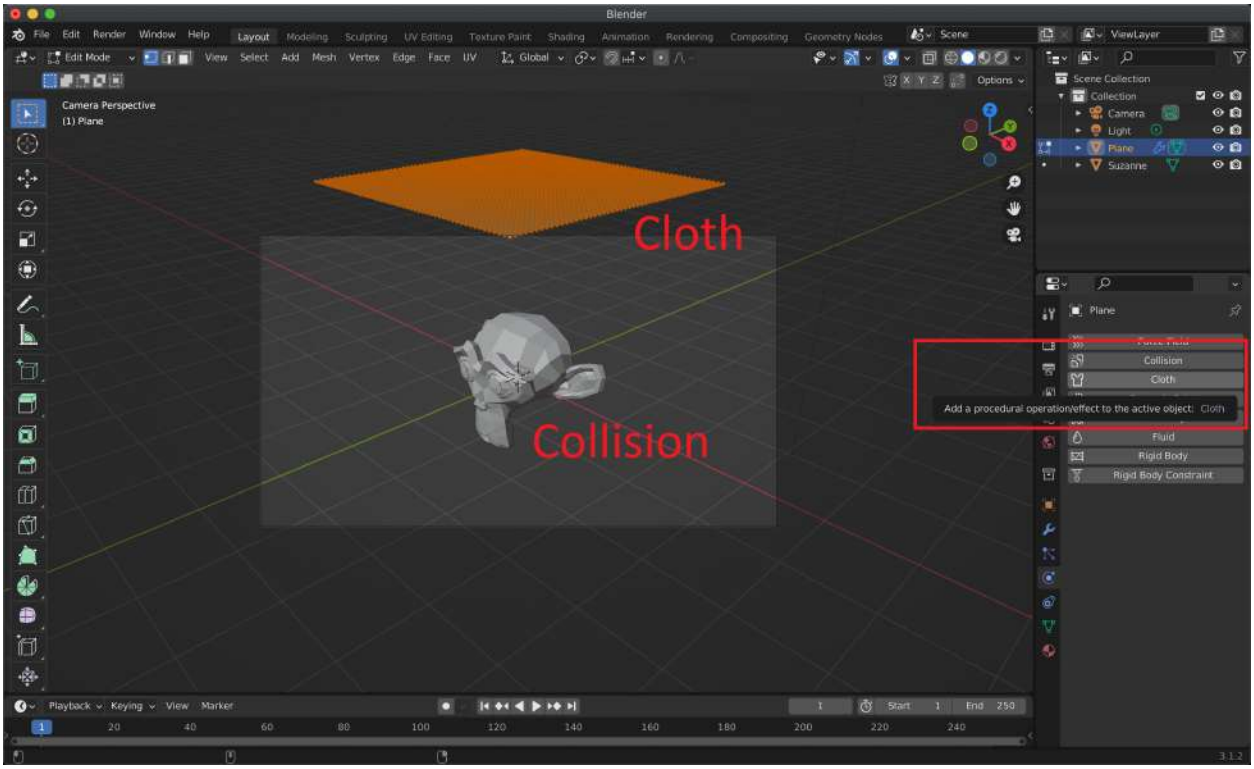


Figure 9

To see your cloth simultion in action, move the **Timeline Editor** at the bottom of the Blender interface up until you see the blue line clearly (see Figure 10). You'll see a timeline starting at 1 and ending at 250 (you can change this to another number like you did with the fluid simulation). You can scrub through the timeline by clicking and dragging the blue line to see the cloth simulation at different frames. For instance, Figure 10 shows us looking at the 31st frame of our cloth simulation.

To get a better sense of how your deformed plane looks like as a cloth, you can turn on **Shade Smooth** when right-clicking it in **Object Mode**. Make sure to change the **Render Engine** to **Cycles** for ray tracing. Then click the render preview on the viewport toolbar (see Figure 10).

When you find a frame in your simulation that you're happy with, you can export all the geometry in that frame as an .obj mesh. Simply use the **File** → **Export** option as usual. Then, you can import that same .obj into your scene and work with it like any other geometry.

Warning: If you try to work off the Blender file from which you created the simulation, then you'll likely run into artifacts from the simulation set up. For instance, the space in which the cloth fell during the simulation may end up getting rendered (which is something you don't want

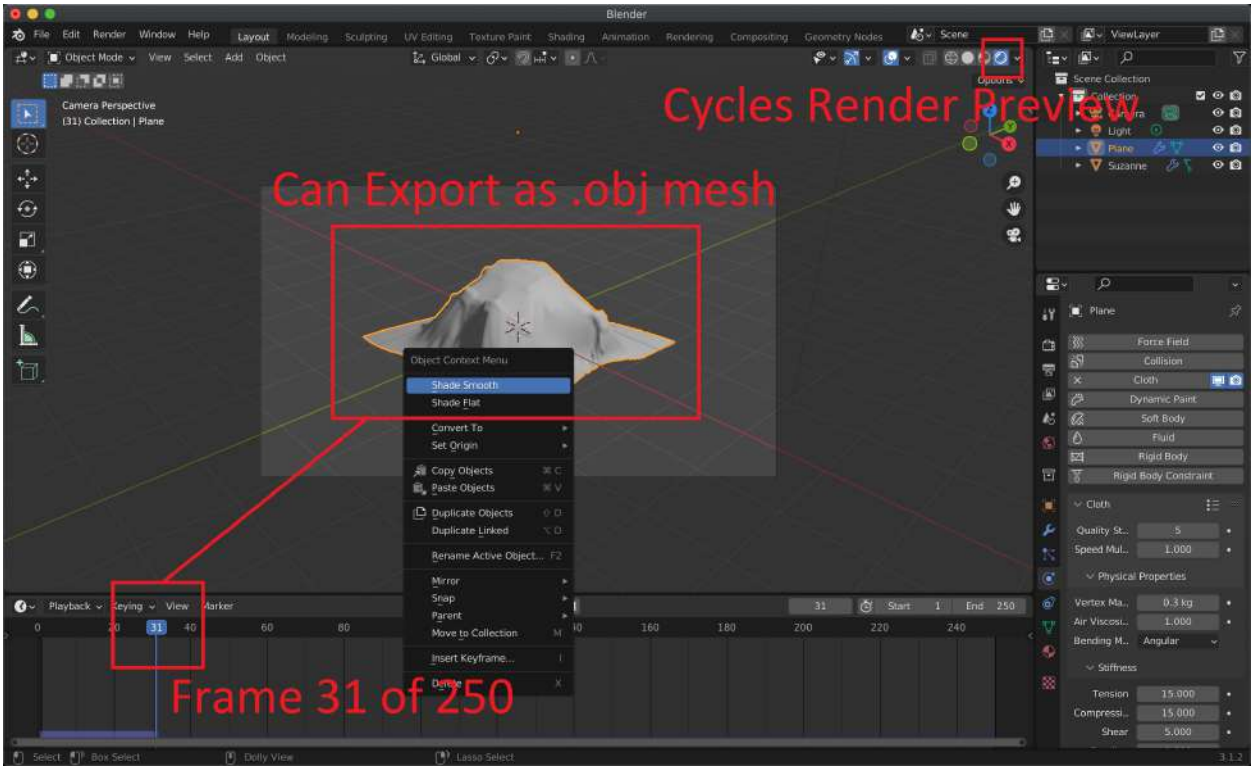
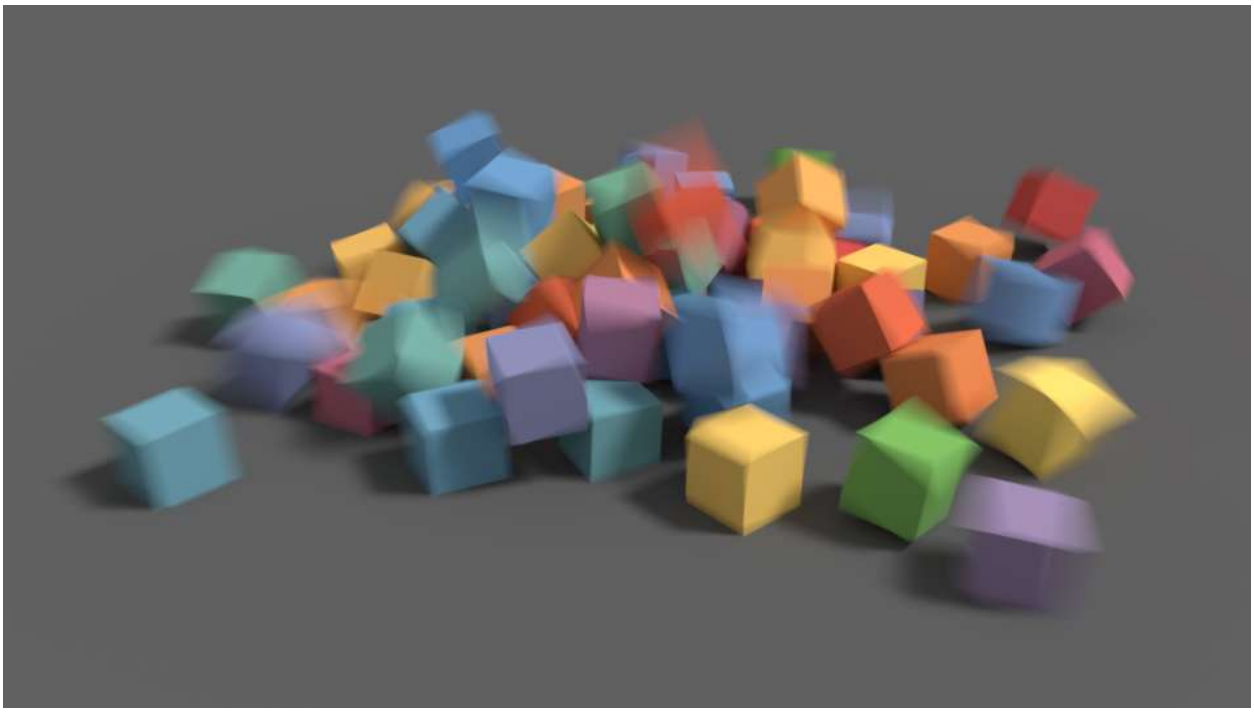


Figure 10

to happen)! So we recommend doing your simulation work in a separate file, and then exporting and importing the result as an .obj mesh to work with in a main scene file.

For more info on the parameters and the various options that Blender provides for cloth simulation, see their official manual. You may also find this quick curtain tutorial useful in combination with this curtain plus wind simulation tutorial.

4 Motion Blur in Blender



Blender provides a very simple way to set up and ray trace motion blur. Let's take a look with a simple example. First, taking the default Blender scene:

- Translate the cube to some location that you want to act as its starting point.
- Move the **Timeline Editor** at the bottom of the Blender interface up until you see the blue line clearly (see Figure 11).
- Right click the cube and click **Insert Keyframe...** (see Figure 11), then click **Location** (see Figure 12). This will mark the current scene as our first position in the timeline and also tell Blender that we would like to interpolate location aka position values for an animation.
- Move the blue line in the **Timeline Editor** to a later frame like frame 10.
- Go back and translate the cube to another location that you want to act its next position.
- Then, using the same steps as above, insert a new location keyframe.

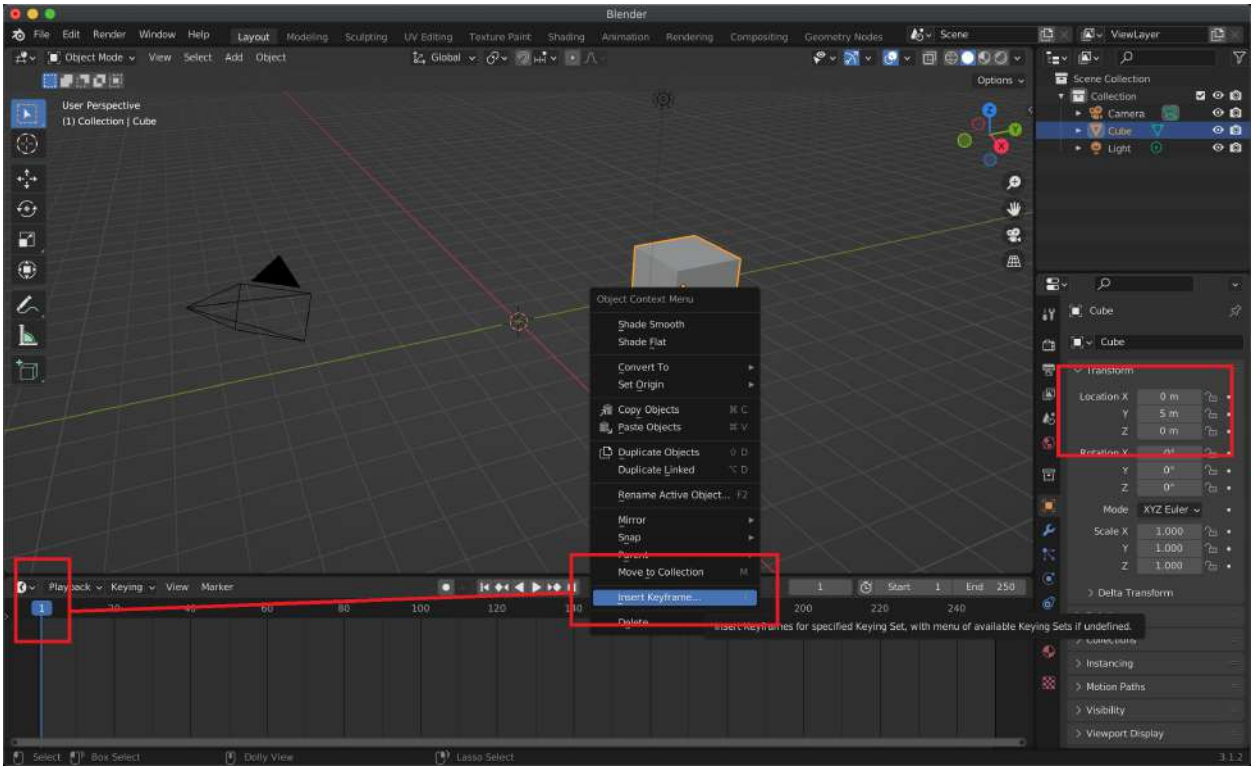


Figure 11

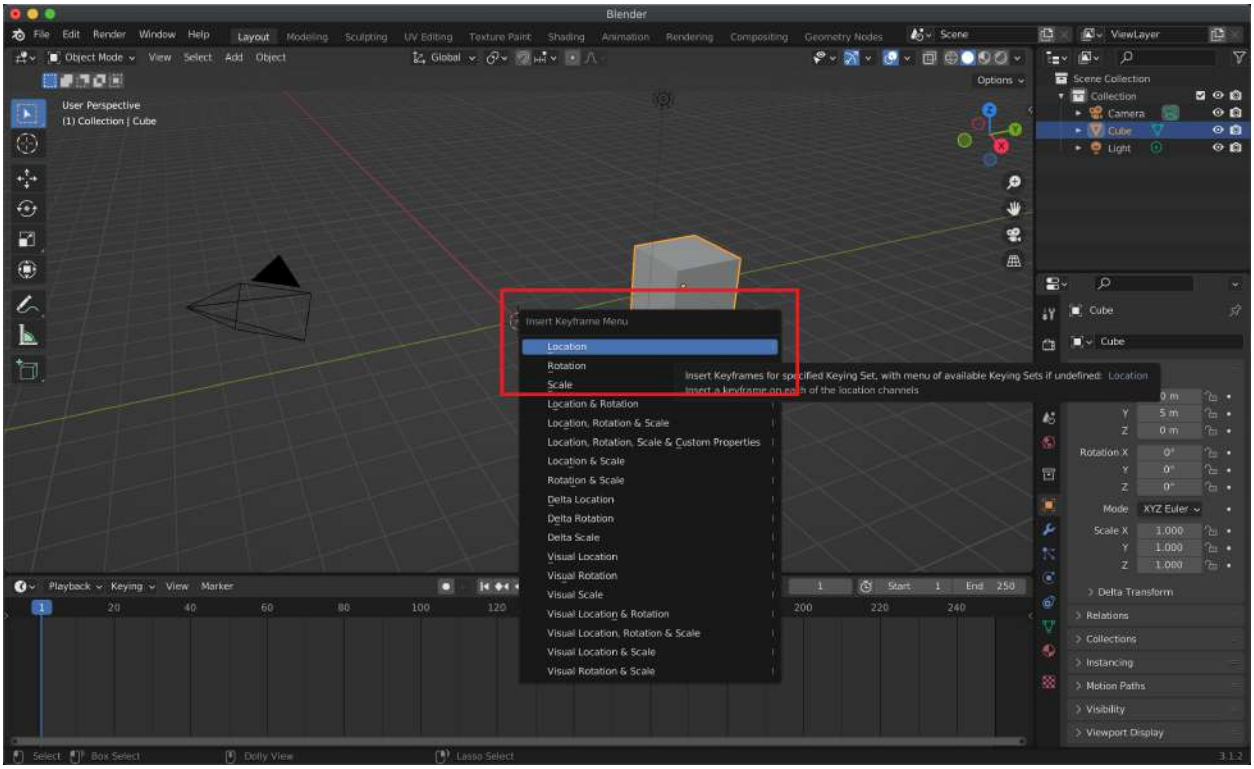


Figure 12

Now, if you scrub through the timeline by clicking and dragging the blue line from frame 1 to frame 10, you'll see your cube move smoothly from its initial position to its next position. Figure 13 shows our example with the cube at frame 7 of the animation and its interpolated position values.

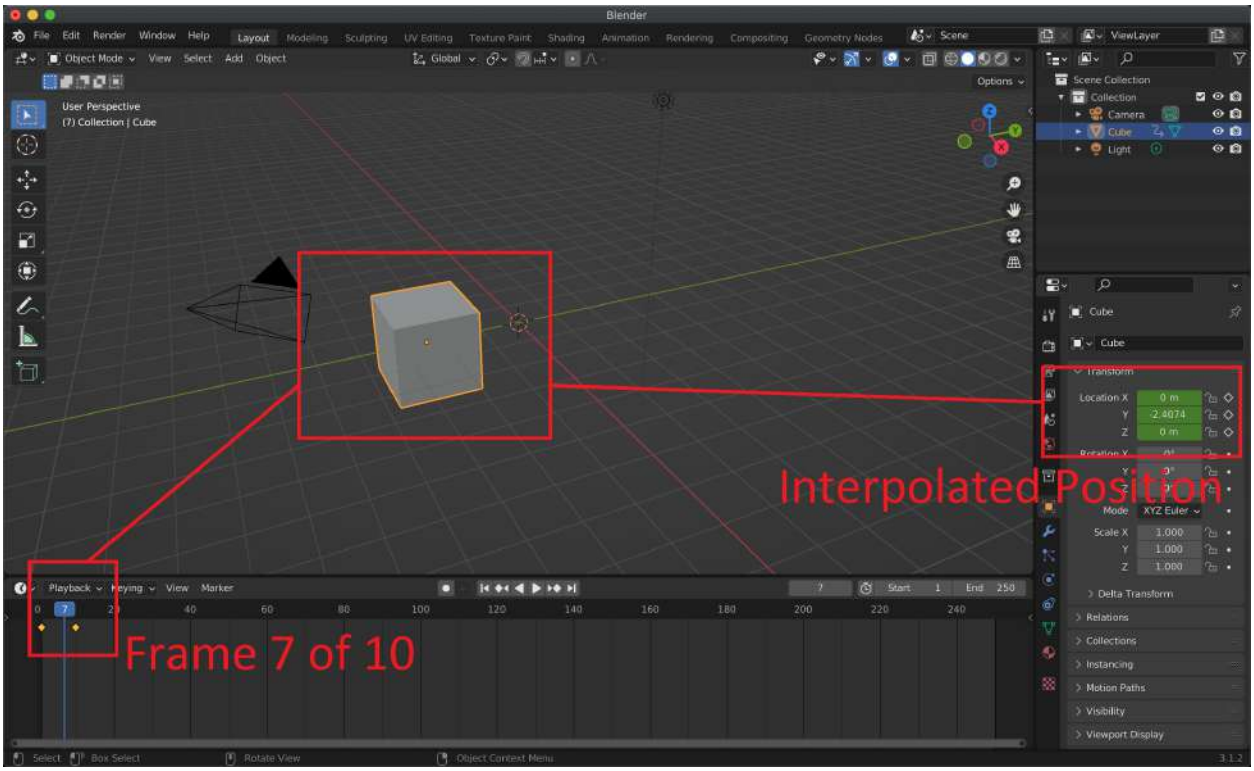
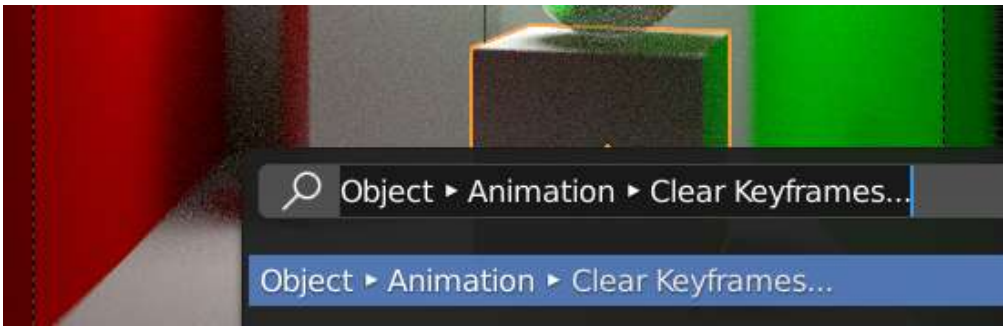


Figure 13

To have Blender ray trace your scene with motion blur, first go to **Render Properties** as usual to change the **Render Engine** to **Cycles** for ray tracing. Then scroll down in this **Properties Editor** tab to find the **Motion Blur** checkbox (see Figure 14). Just check the box, and Blender will do motion blur! To see, pick a frame in the middle of the animation (e.g. frame 7 of 10 in the timeline) and render your scene as an image. You should see something like that in Figure 14 pop up in your Blender render window.

If you ever want to remove all the animation data that you created for the motion blur, then you can do so by selecting all the keyframes in the **Timeline Editor** and deleting them (right-click → **Delete Keyframes**). Alternatively, you can select your motion blurred object (e.g. the cube) and press **F3** to pull up the **Menu Search**, then type **Clear Keyframes**.



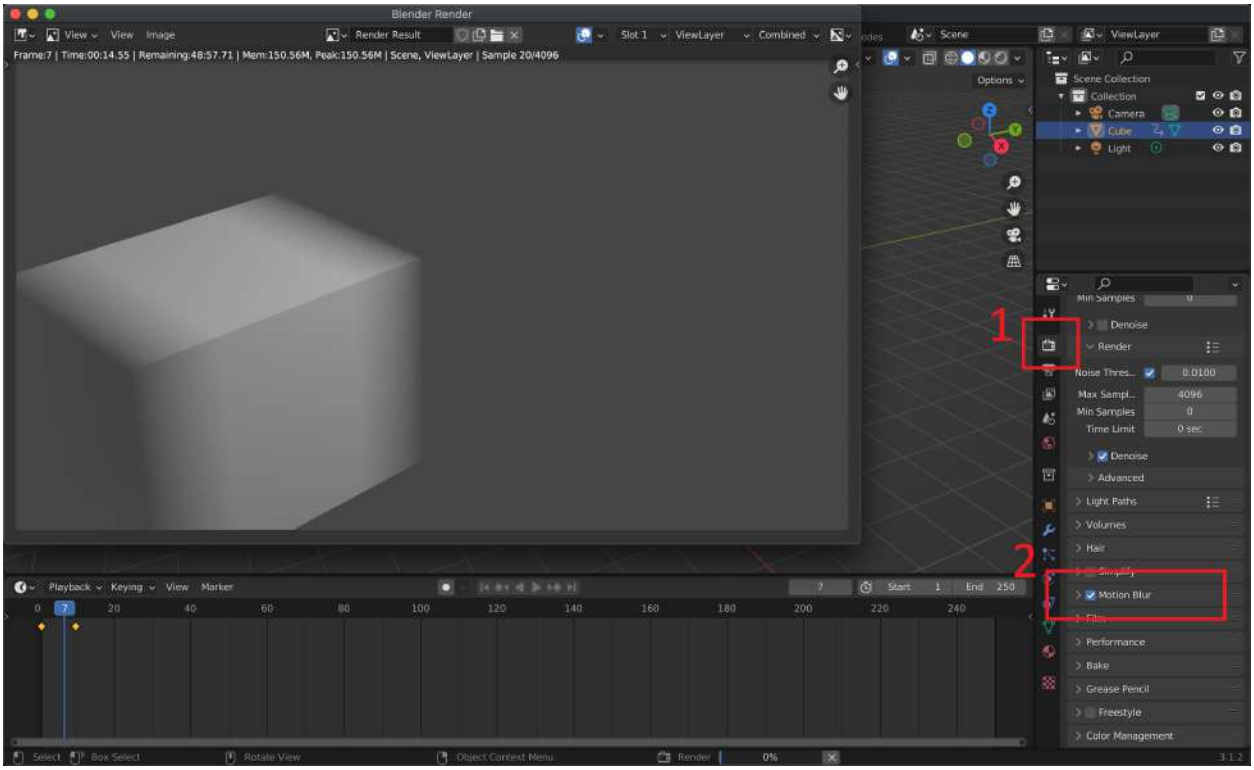
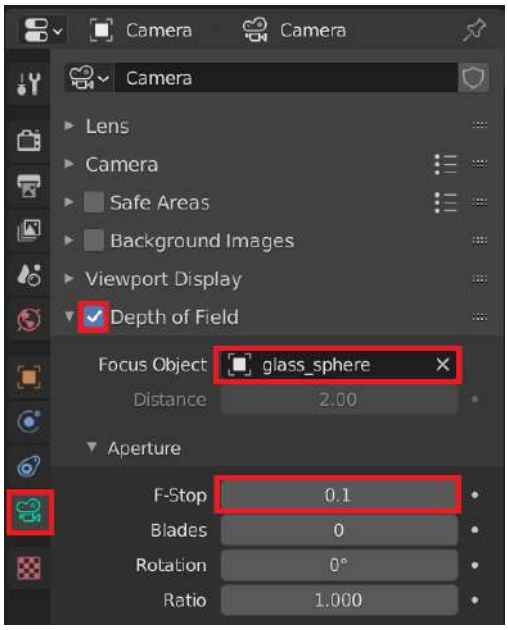


Figure 14

5 Depth of Field in Blender

Blender provides a very simple way to set up and ray trace depth of field. Let's take a look with a simple example. Open this .blend file containing the Cornell Box scene. Select the **Camera** in the **Scene Collection** in the upper-right of the interface. Then go to the camera's **Object Data Properties** in the **Properties Editor**. Click the checkbox for **Depth of Field**, and set the **Focus Object** to the **glass_sphere**. This will make our depth of field focus on the right-most sphere.



Play around with different values of **F-Stop** to get a feel for how it affects the strength of the blur. Lower values will increase blur, while higher values make the image more clear overall. For instance, try rendering the Cornell Box scene with depth of field focusing on the right-most, glass sphere using a **F-Stop** of 0.1. Then compare that sphere visually to the other sphere.

6 Volume Rendering in Blender

Volume rendering is an advanced technique for achieving effects that cannot be represented by surface meshes alone. More specifically, volume rendering is designed to render volumetric objects like smoke, fire, fog, and clouds. We will try out Blender's volume rendering capabilities with a simple fog example. For more info on all the options that Blender has to offer for volume rendering, see its manual on volumetric effects (It's for an older version of Blender, but should still be relevant).

To set up our fog example, create a new default scene and enlarge our default cube. Then:

- Go to **Material Properties** in the Properties Editor (#1 in Figure 15).
- Click on the **Surface** option (#2 in Figure 15), then **Remove** or **Disconnect** (#3 in Figure 15) the default Principled BSDF material.
- Scroll down to the **Volume** options (#1 in Figure 16).
- Add a **Principled Volume** in the **Volume** option.
- Then, set the **Density** (of our fog) to 0.2 (#2 in Figure 16)
- Add another cube inside our enlarged cube. You may want to switch to wireframe view when doing this.

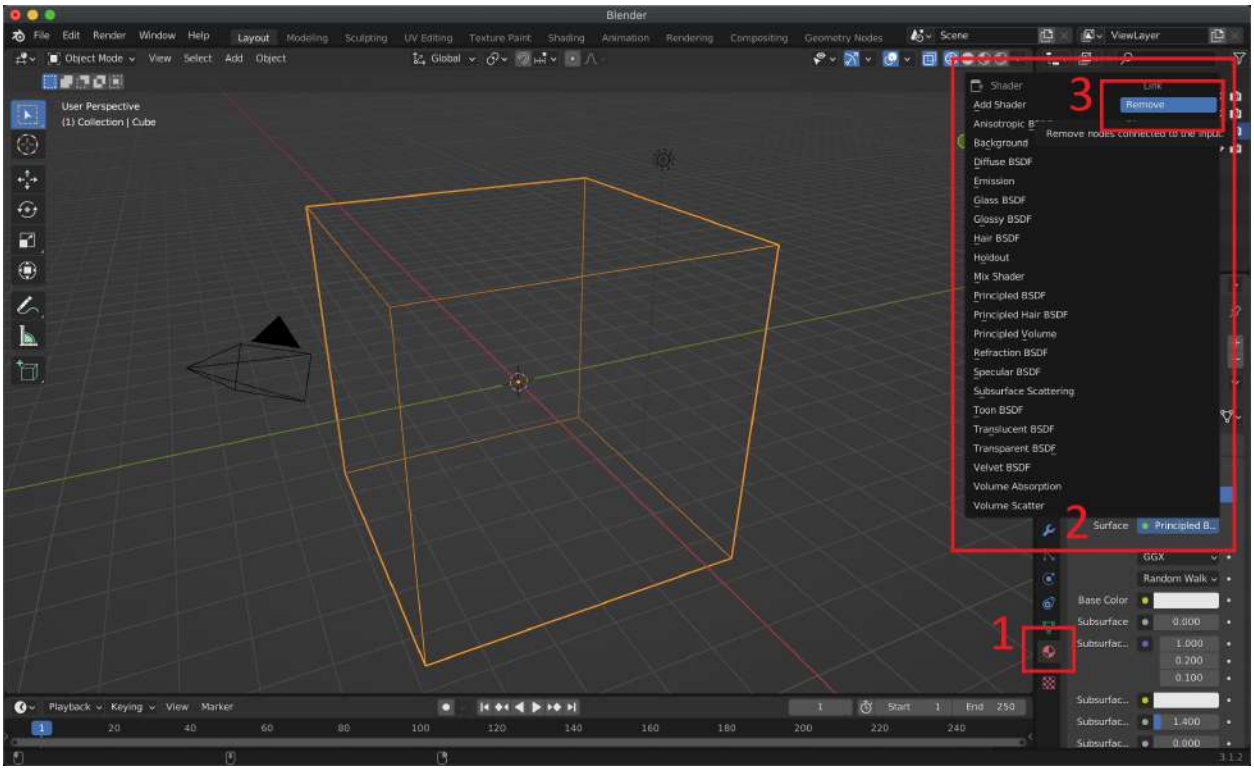


Figure 15

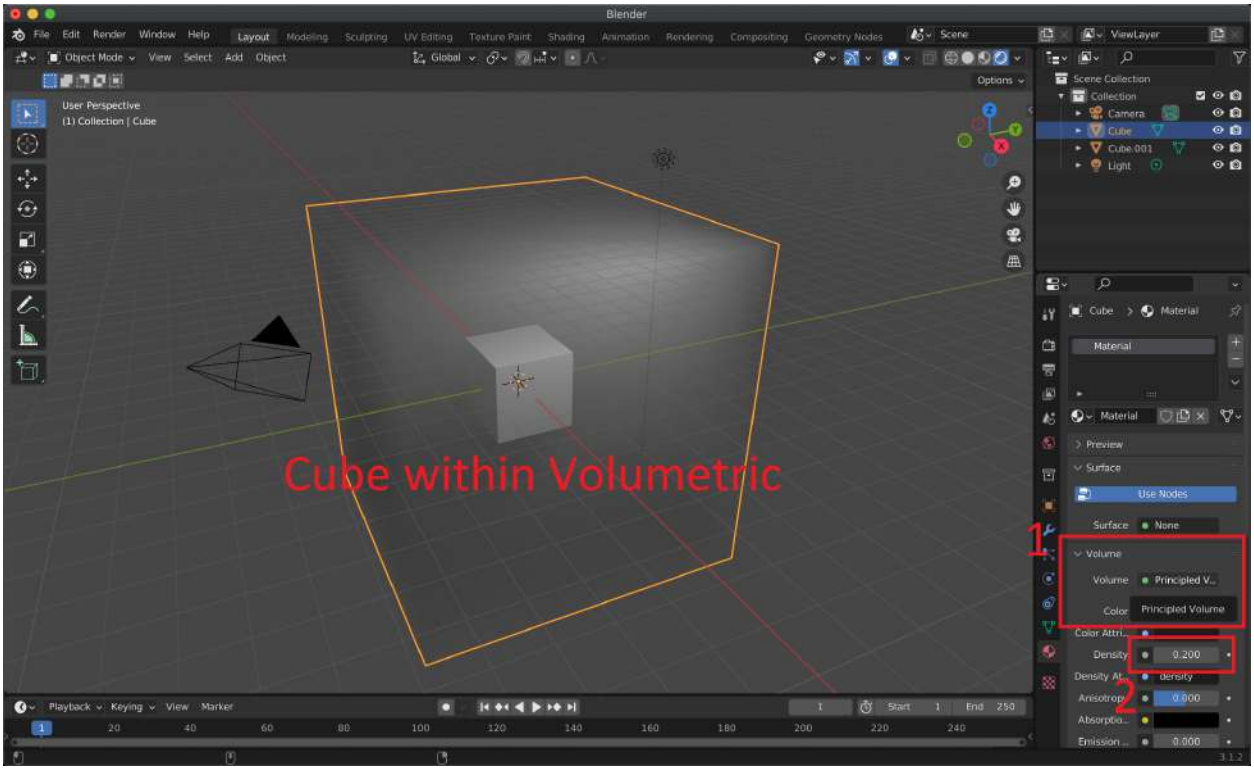


Figure 16

Set Blender to **Cycles** and toggle on the render preview. You should see a fog like effect inside the outer cube surrounding your inner cube. Render the image to see something similar to Figure 17.

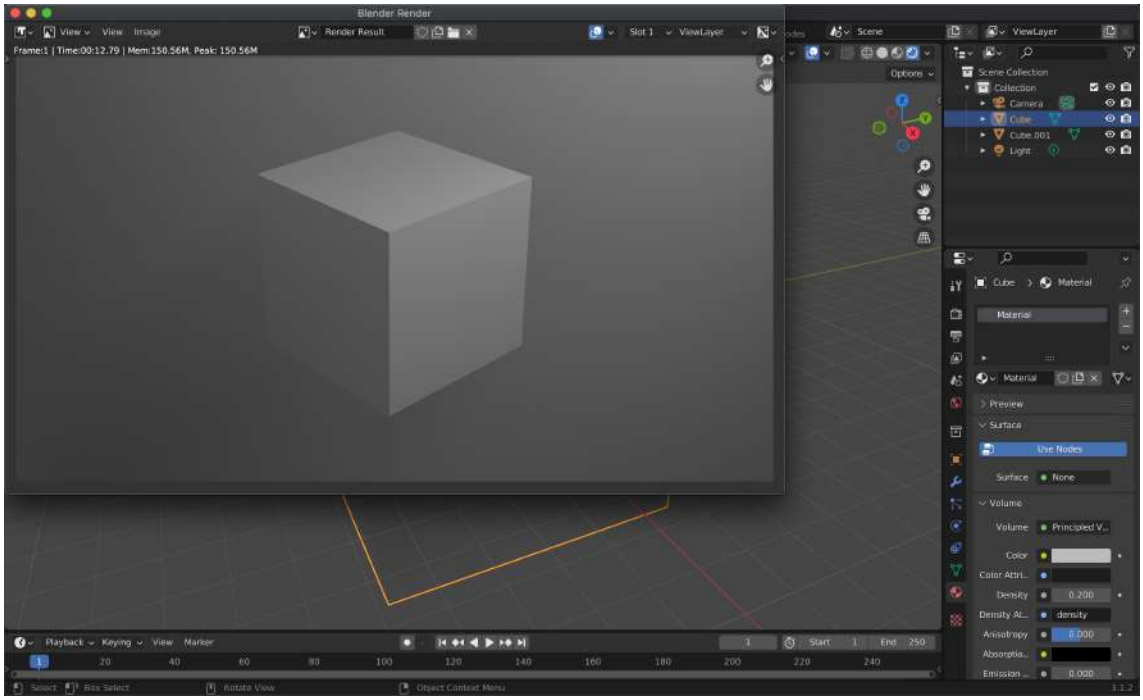


Figure 17

Try adding a fog effect to the Cornell Box scene from Section 5 (on depth of field) – it results in a pretty interesting visual phenomena. First, we need to surround the entire Cornell Box scene with a volumetric cube. You can do this by adding a new cube, translating it to $(-2.75, 2.75, 2.75)$, and scaling all its dimensions by 4. When adding the material, you will have to create a **New** material first. When you render the Cornell Box scene with fog, you might notice that the shape of the light becomes visible for reasons we will discuss later in lecture!